

بسم الله الرحمن الرحيم

درخت های قرمز-سیاه - Trees Red-Black

مرداد ۱۴۰۴

فهرست مطالب

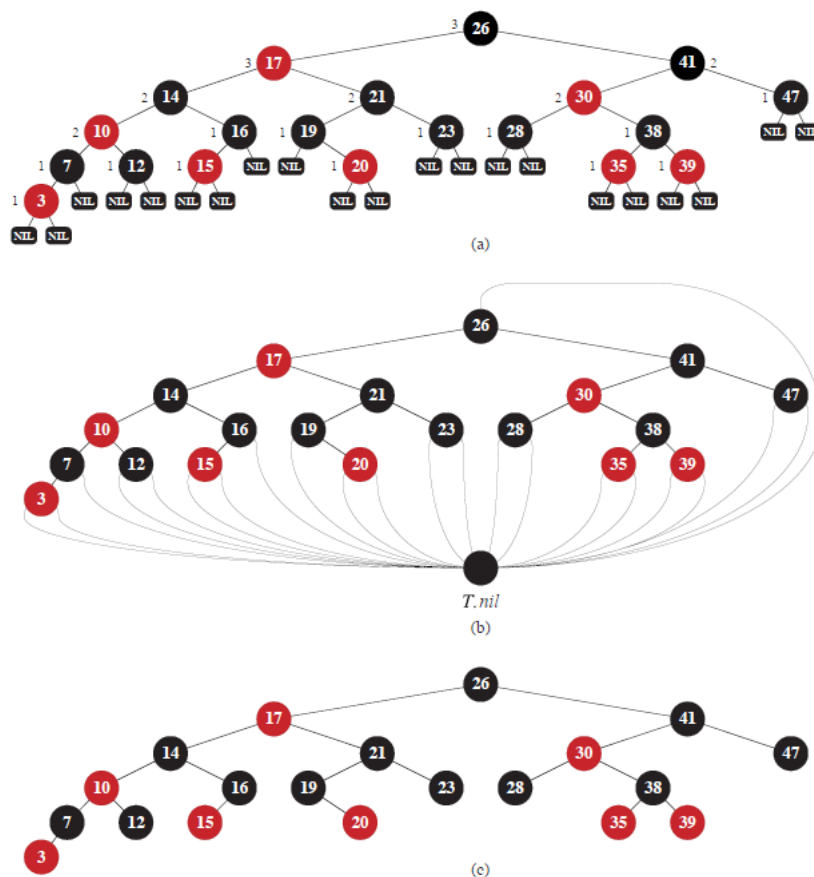
۳	۱	مقدمه
۳	۲	تعریف درخت قرمز-سیاه
۴	۳	ویژگی‌های درخت قرمز-سیاه
۵	۴	ارتفاع درخت قرمز-سیاه
۵	۵	چرخش در درخت قرمز-سیاه
۵	۱.۵	LEFT-ROTATE
۷	۲.۵	RIGHT-ROTATE
۸	۶	درج در درخت قرمز-سیاه
۱۲	۱.۶	FIXUP درج در درخت قرمز-سیاه
۱۴	۷	حذف در درخت قرمز-سیاه
۱۵	۱.۷	RB-DELETE-FIXUP
۱۶	۲.۷	RB-DELETE-FIXUP مثال پیشرفته
۲۱	۸	منابع

۱ مقدمه

Red-black و یا درخت های قرمز و سیاه درخت های جستجوی دودویی هستند که هر گره آنها دارای رنگ قرمز و یا مشکی است. درخت red-black دارای ارتفاع $\log n$ است ، بنابراین عملیات های درج، جستجو و حذف در آن در بدترین حالت $O(\log n)$ زمان می‌برند.

۲ تعریف درخت قرمز-سیاه

درخت قرمز-سیاه نوعی Binary Search Tree است که در آن به هر گره یک رنگ (قرمز یا سیاه) نسبت داده می‌شود. قوانین رنگ‌بندی و چرخش‌ها باعث حفظ توازن درخت در حین عملیات درج و حذف می‌شوند.



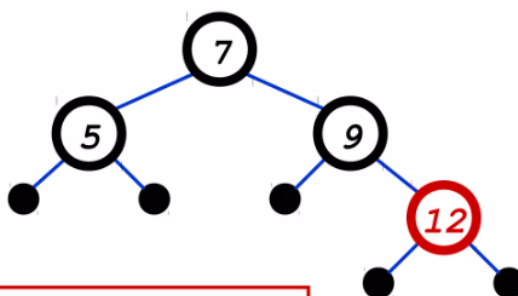
شکل ۱: نمونه تصویر درخت سیاه-قرمز x

۳ ویژگی‌های درخت قرمز-سیاه

۱. هر گره یا قرمز است یا سیاه.
۲. ریشه همیشه سیاه است.
۳. همه ی برگ‌ها (گره‌های NIL یا تهی) سیاه هستند.
۴. اگر گره‌ای قرمز باشد، فرزندان آن حتماً سیاه هستند.
۵. تعداد گره‌های سیاه در هر مسیر از یک گره تا برگ‌هایش باید یکسان باشد.

Red-Black Trees: An Example

- *Color this tree:*



Red-black properties:

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

شکل ۲: نمونه درخت قرمز-مشکی

۴ ارتفاع درخت قرمز-سیاه

حداکثر ارتفاع

• حداقل طول درخت Red-Black برابر با **black-height** یا ارتفاع گره‌های مشکی است.

• حداقل ارتفاع h برای n گره برابر است با:

$$h \geq \lfloor \log_2(n+1) \rfloor$$

این حالت زمانی رخ می‌دهد که درخت Red-Black به صورت یک درخت دودویی کامل باشد، یعنی:

$$n = 2^h - 1$$

• در بدترین حالت، ارتفاع Red-Black Tree از رابطه زیر پیروی می‌کند:

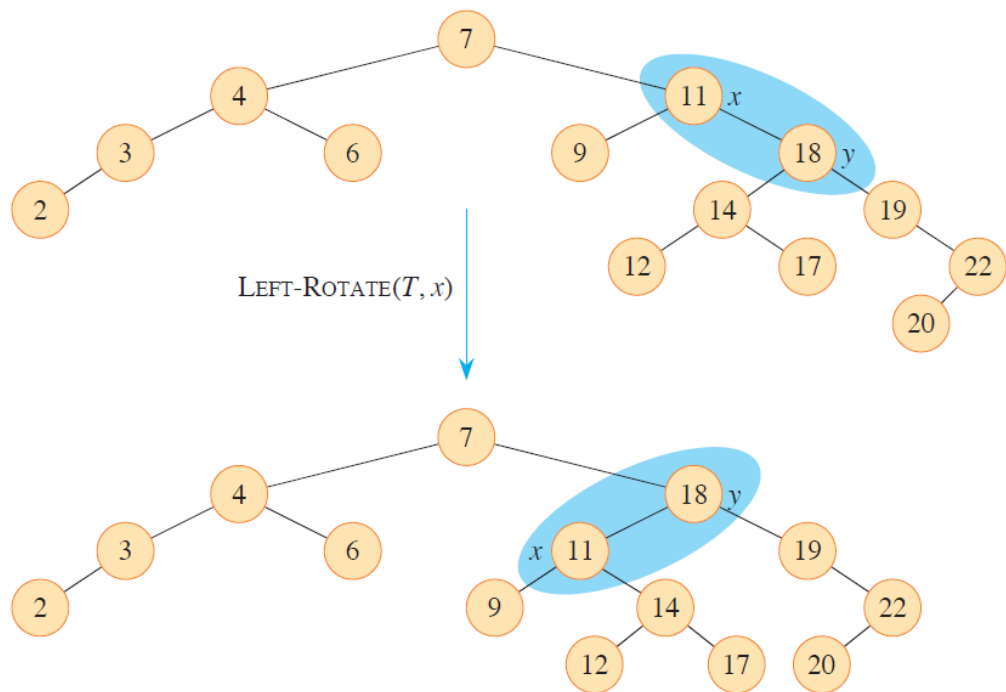
$$h \leq 2 \log_2(n+1)$$

۵ چرخش در درخت قرمز-سیاه

چرخش‌ها عملیات اصلی برای حفظ ویژگی‌های درخت قرمز-سیاه در حین درج و حذف هستند. این عملیات به صورت محلی ساختار درخت را تغییر می‌دهند اما خاصیت درخت جستجوی دودویی را حفظ می‌کنند و در زمان $O(1)$ انجام می‌شوند. دو نوع چرخش وجود دارد: چرخش به چپ و چرخش به راست.

LEFT-ROTATE ۱.۵

در چرخش به چپ حول گره x ، فرض می‌شود که فرزند راست آن یعنی y تهی نیست. این چرخش باعث می‌شود y به جای x قرار گیرد و x به عنوان فرزند چپ y درآید. زیردرخت چپ y نیز به عنوان زیردرخت راست x متصل می‌شود.



شکل ۳: عملکرد چرخش به چپ حول گره x

```

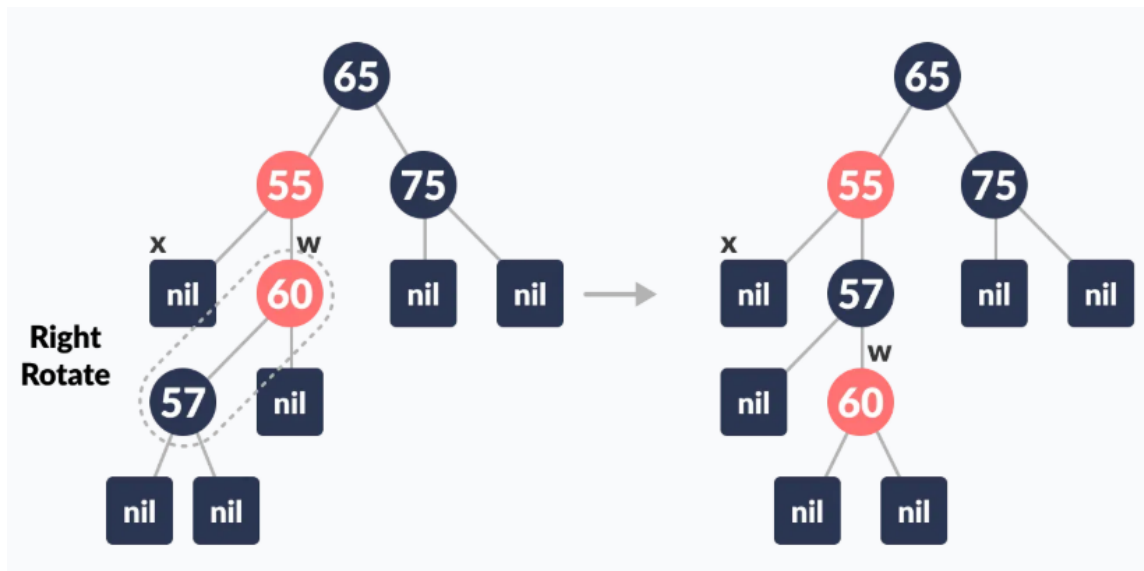
۱ LEFT-ROTATE(T, x):
۲   y = x.right
۳   x.right = y.left
۴   if y.left != T.nil:
۵       y.left.p = x
۶   y.p = x.p
۷   if x.p == T.nil:
۸       T.root = y
۹   else if x == x.p.left:
۱۰       x.p.left = y
۱۱   else:
۱۲       x.p.right = y
۱۳   y.left = x
۱۴   x.p = y

```

Listing ۱: Left Rotation Pseudocode

RIGHT-ROTATE ۲.۵

چرخش به راست عملیات معکوس چرخش به چپ است. در این چرخش حول گره y ، فرض می‌شود که فرزند چپ آن یعنی x تهی نیست.



شکل ۴: عملکرد چرخش به راست حول گره $y=w$

```

1  RIGHT-ROTATE(T, y):
2      x = y.left
3      y.left = x.right
4      if x.right != T.nil:
5          x.right.p = y
6      x.p = y.p
7      if y.p == T.nil:
8          T.root = x
9      else if y == y.p.right:
10         y.p.right = x
11     else:
12         y.p.left = x
13     x.right = y
14     y.p = x

```

Listing ۲: Right Rotation Pseudocode

۶ درج در درخت قرمز-سیاه

درج گره جدید ممکن است باعث نقض قوانین درخت شود و ساختار آن را به هم بزند. در این شرایط، برای حفظ خاصیت‌های درخت، از چرخش‌ها و رنگ‌آمیزی مجدد استفاده می‌شود. این اصلاحات باعث می‌شوند که درخت به حالت متعادل و درست بازگردد. هدف اصلی این است که خاصیت‌های اساسی درخت مانند تعادل رنگ‌ها و ترتیب گره‌ها حفظ شود. بر اساس وضعیت درخت پس از درج گره، سه حالت مختلف ممکن است پیش بیاید. هر یک از این حالات نیازمند راه‌حل خاص خود است که با چرخش‌ها و تغییر رنگ‌ها اصلاح می‌شود. این فرایندها به صورت خودکار و مرحله به مرحله انجام می‌گیرند. در ادامه، هر یک از این حالت‌ها به صورت دقیق و گام به گام بررسی خواهند شد.

کد الگوریتم درج

```
۱ RB-Insert(T, z):
۲   y = NIL
۳   x = T.root
۴   while x != NIL:
۵       y = x
۶       if z.key < x.key:
۷           x = x.left
۸       else:
۹           x = x.right
۱۰  z.p = y
۱۱  if y == NIL:
۱۲      T.root = z
۱۳  else if z.key < y.key:
۱۴      y.left = z
۱۵  else:
۱۶      y.right = z
۱۷  z.left = NIL
۱۸  z.right = NIL
۱۹  z.color = RED
۲۰  RB-Insert-Fixup(T, z)
```

Listing :۳ Red-Black Tree Insertion

حالت ۱: عموی گره جدید (y) قرمز است

در این حالت، عموی گره x (که تازه درج شده) قرمز رنگ است. این وضعیت ساده‌ترین حالت است و با تغییر رنگ‌های مربوطه حل می‌شود.

۱. گره والد p و عموی y گره x را به رنگ مشکی در می‌آوریم.

۲. گره پدر بزرگ p-p را به رنگ قرمز در می‌آوریم.

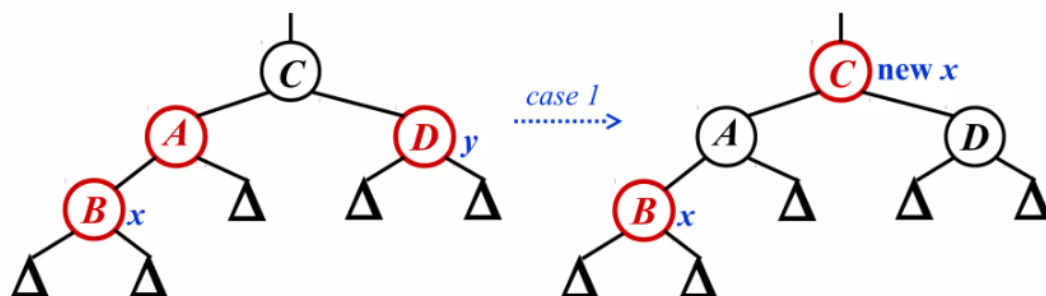
۳. اشاره‌گر x را به گره پدر بزرگ منتقل می‌کنیم تا در تکرار بعدی حلقه (در صورت لزوم) بررسی‌ها از آنجا ادامه پیدا کند.

این عملیات به گونه‌ای انجام می‌شود که تعداد گره‌های مشکی در مسیرهای مختلف درخت حفظ شود.

RB Insert: Case 1

```
if (y->color == RED)
    x->p->color = BLACK;
    y->color = BLACK;
    x->p->p->color = RED;
    x = x->p->p;
```

- Case 1: "uncle" is red
- In figures below, all Δ 's are equal-black-height subtrees



Same action whether x is a left or a right child

شکل ۵: درج در درخت قرمز-مشکی: حالت ۱

حالت ۲: عموی گره جدید (y) مشکلی است و گره x فرزند راست است

در این حالت، عموی گره x مشکلی رنگ است و گره x یک فرزند راست برای والد خود p است. برای حل این وضعیت، ابتدا باید این حالت را به حالت ۳ تبدیل کنیم.

۱. یک چرخش به چپ (left rotation) روی گره x انجام می‌دهیم.

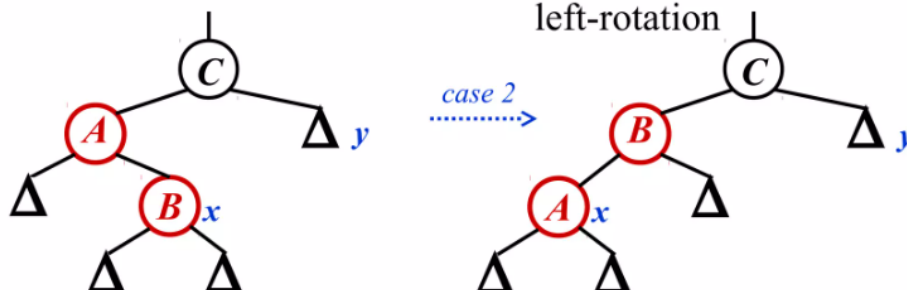
۲. پس از چرخش، وضعیت گره‌ها به گونه‌ای می‌شود که می‌توانیم آن را به عنوان حالت ۳ در نظر بگیریم و ادامه عملیات را طبق آن حالت پیش ببریم.

این تبدیل اطمینان حاصل می‌کند که خاصیت ۴ (تمام مسیرهای پایین‌رونده حاوی تعداد یکسانی از گره‌های مشکلی هستند) حفظ شود.

RB Insert: Case 2

```
if (x == x->p->right)
    x = x->p;
    leftRotate(x);
// continue with case 3 code
```

- Case 2:
 - “Uncle” is black
 - Node x is a right child
- Transform to case 3 via a left-rotation



Transform case 2 into case 3 (x is left child) with a left rotation

This preserves property 4: all downward paths contain same number of black nodes

شکل ۶: درج در درخت قرمز-مشکی: حالت ۲

در ادامه حالت سوم را بررسی می‌کنیم :

حالت ۳: عموی گره جدید (y) مشکلی است و گره x فرزند چپ است

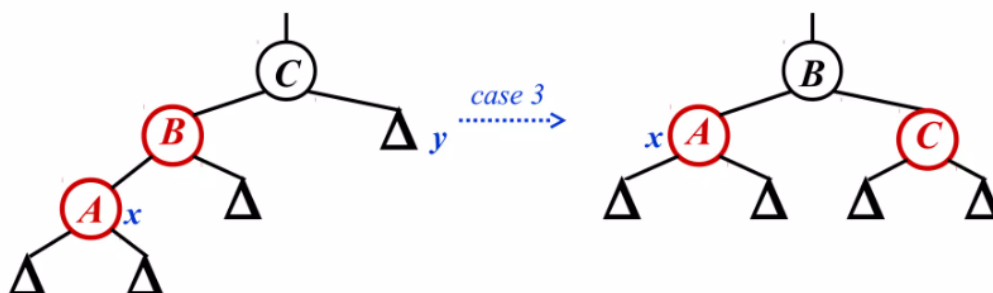
این حالت معمولاً پس از حالت ۲ (در صورت نیاز به تبدیل) یا به صورت مستقیم رخ می‌دهد. در این وضعیت، عموی گره x مشکلی است و گره x یک فرزند چپ برای والد خود p است.

۱. رنگ گره والد p را به مشکلی تغییر می‌دهیم.
 ۲. رنگ گره پدر بزرگ $p \rightarrow p-$ را به قرمز تغییر می‌دهیم.
 ۳. یک چرخش به راست (right rotation) روی گره پدر بزرگ $p \rightarrow p-$ انجام می‌دهیم.
- این عملیات با حفظ خاصیت‌های درخت قرمز-مشکی، تعادل را برقرار می‌کند. شکل زیر حالت سوم اضافه کردن گره را نشان می‌دهد:

RB Insert: Case 3

```
x->p->color = BLACK;
x->p->p->color = RED;
rightRotate(x->p->p);
```

- Case 3:
 - "Uncle" is black
 - Node x is a left child
- Change colors; rotate right



Perform some color changes and do a right rotation
Again, preserves property 4: all downward paths contain same number of black nodes

شکل ۷: درخت قرمز-مشکی: حالت ۳

۱.۶ FIXUP درج در درخت قرمز-سیاه

در هنگام درج گره، ممکن است خاصیت‌های درخت قرمز-مشکی نقض شود که با اجرای الگوریتم Fixup و اعمال چرخش و رنگ‌آمیزی، تعادل آن بازگردانده می‌شود.

```

۱  RB-INSERT-FIXUP(T, z):
۲      while z.p.color == RED
۳          if z.p == z.p.p.left // is z's parent a
                left child?
۴              y = z.p.p.right // is z's uncle
۵              if y.color == RED // are z's parent and
                uncle both red?
۶                  z.p.color = BLACK
۷                  y.color = BLACK
۸                  z.p.p.color = RED
۹                  z = z.p.p // case 1
۱۰             else // uncle is black
۱۱                 if z == z.p.right // case 2
۱۲                     z = z.p
۱۳                     LEFT-ROTATE(T, z)
۱۴                 // case 3
۱۵                 z.p.color = BLACK
۱۶                 z.p.p.color = RED
۱۷                 RIGHT-ROTATE(T, z.p.p)
۱۸             else // same as lines 3-15, but with
                "right" and "left" exchanged
۱۹                 y = z.p.p.left
۲۰                 if y.color == RED
۲۱                     z.p.color = BLACK
۲۲                     y.color = BLACK
۲۳                     z.p.p.color = RED
۲۴                     z = z.p.p
۲۵                 else
۲۶                     if z == z.p.left
۲۷                         z = z.p
۲۸                         RIGHT-ROTATE(T, z)
۲۹                     z.p.color = BLACK
۳۰                     z.p.p.color = RED
۳۱                     LEFT-ROTATE(T, z.p.p)
۳۲             T.root.color = BLACK
```

Listing :۴ RB-INSERT-FIXUP

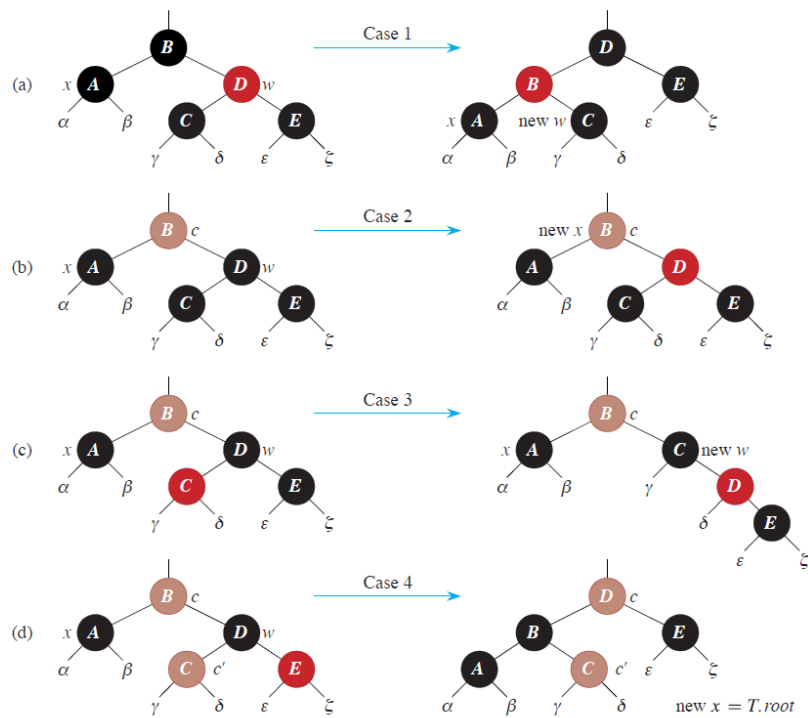
Insert 10, 20, 30 and 15 in an empty tree

The diagram illustrates the insertion of nodes 10, 20, 30, and 15 into a B+ tree. The process is shown in several steps:

- Initial Insertions:** Node 10 is inserted (Case 3, x is root). Then node 20 is inserted (p is parent of x). Then node 30 is inserted (g is grandparent of x).
- Right-Right Case:** An arrow labeled "Case 2 (b) iii Right Right" points to a tree where node 20 is the root, with children 10 and 30. Node 30 is marked with 'x'.
- Insert 15:** An arrow labeled "Insert 15" points down to a new tree structure.
- Left-Right Case:** The tree has root 20 with children 10 and 30. Node 15 is the left child of 10. Node 15 is marked with 'x'.
- Left-Left Case:** An arrow labeled "Case 3 (x is root)" points to a tree where node 20 is the root, with children 10 and 30. Node 15 is the left child of 10. Node 15 is marked with 'x'.
- Right-Left Case:** An arrow labeled "Case 2(a)" points to a tree where node 20 is the root, with children 10 and 30. Node 15 is the right child of 10. Node 15 is marked with 'x'.

Note: NULL is considered as Black

شکل ۸: مثال اجرای Fixup بعد از هر بار insert کردن



شکل ۹: مثال اجرای Fixup بعد از هر بار insert کردن

۷ حذف در درخت قرمز-سیاه

حذف در درخت قرمز-مشکی پیچیده‌تر از درج است، زیرا ممکن است تعادل گره‌های مشکی یا قوانین رنگ نقض شود. الگوریتم RB-DELETE گره را با جانشین مناسب جایگزین کرده و در صورت نیاز، با اجرای RB-DELETE-FIXUP توازن درخت را بازمی‌گرداند. در ادامه، این الگوریتم را بررسی می‌کنیم.

```

۱  RB-DELETE(T, z)
۲      y = z
۳      y-original-color = y.color
۴      if z.left == T.nil
۵          x = z.right
۶          RB-TRANSPLANT(T, z, z.right)
           // replace z by its right child
۷      elseif z.right == T.nil
۸          x = z.left
۹          RB-TRANSPLANT(T, z, z.left)
           // replace z by its left child
۱۰     else
۱۱         y = TREE-MINIMUM(z.right)
           // y is z's successor
۱۲         y-original-color = y.color
۱۳         x = y.right
۱۴         if y == z.right
۱۵             RB-TRANSPLANT(T, y, y.right)
                 // replace y by its right child
۱۶             y.right = z.right
۱۷             y.right.p = y
۱۸         else
۱۹             x.p = y
           // in case x is T.nil
۲۰         RB-TRANSPLANT(T, z, y)
           // replace z by y
۲۱         y.left = z.left
           // and give z's left child to y
۲۲         y.left.p = y
۲۳         y.color = z.color
۲۴         if y-original-color == BLACK
۲۵             RB-DELETE-FIXUP(T, x)
                 // correct any violations
```

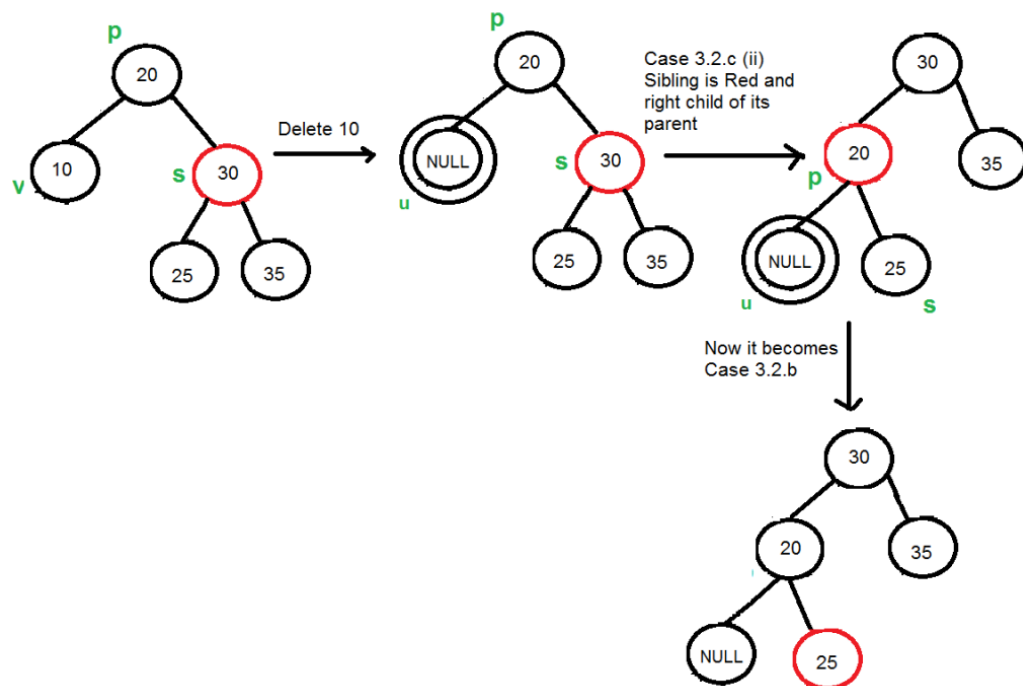
Listing :۵ Red-Black Tree Deletion

1.7 RB-DELETE-FIXUP

پس از حذف یک گره در درخت قرمز-مشکی، اگر گره حذف شده یا جانشین آن مشکی باشد، ممکن است قوانین قرمز-مشکی نقض شوند. الگوریتم RB-DELETE-FIXUP برای بازگرداندن این خاصیت‌ها اجرا می‌شود. این الگوریتم با پیمایش از گره x به سمت ریشه، با چرخش‌ها و تغییر رنگ‌ها تعادل درخت را بازمی‌گرداند.

```
1  RB-DELETE-FIXUP(T, x)
2  while x != T.root and x.color == BLACK
3  if x == x.p.left // case1
4      w = x.p.right
5      if w.color == RED
6          w.color = BLACK
7          x.p.color = RED
8          LEFT-ROTATE(T, x.p)
9          w = x.p.right
10     if w.left.color == BLACK and
11         w.right.color == BLACK // case2
12         w.color = RED
13         x = x.p
14     else
15         if w.right.color == BLACK // case3
16             w.left.color = BLACK
17             w.color = RED
18             RIGHT-ROTATE(T, w)
19             w = x.p.right
20             w.color = x.p.color // case4
21             x.p.color = BLACK
22             w.right.color = BLACK
23             LEFT-ROTATE(T, x.p)
24             x = T.root
25     else
26         (*@ // same as then-clause with
27             "right" and "left" exchanged @*)
```

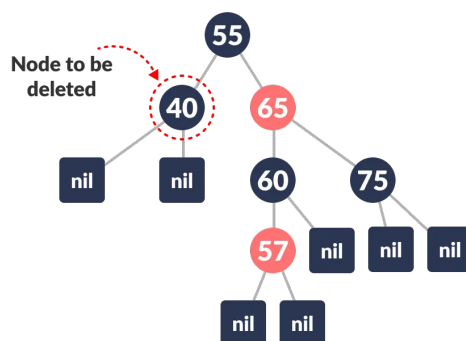
Listing ۶: Red-Black Tree Deletion



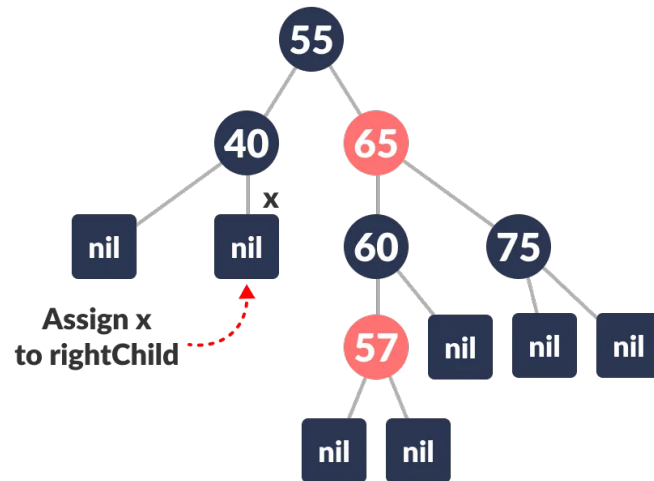
شکل ۱۰: مثال اجرای Fixup بعد از هر بار Delete کردن

۲.۷ RB-DELETE-FIXUP مثال پیشرفته

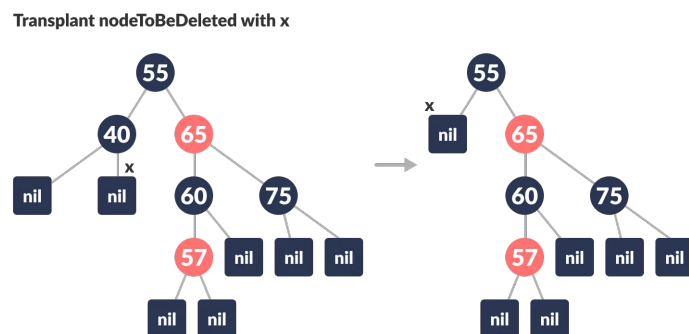
در ادامه یک مثال پیشرفته و چند مرحله ای از حذف گره را تا زمان برقراری کامل تعادل بررسی می کنیم :



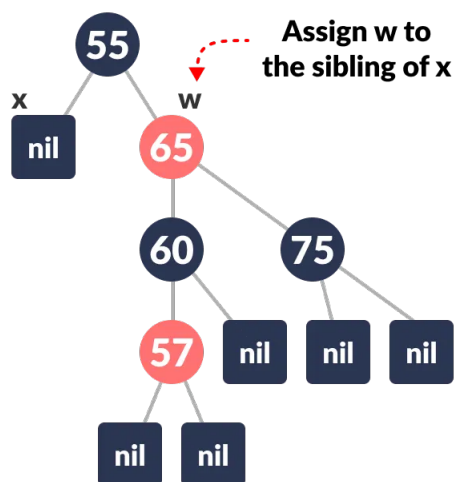
شکل ۱۱: شناسایی گره ۴۰ برای حذف؛ گره ۴۰ یک گره قرمز بدون فرزند است که به سادگی حذف می‌شود.



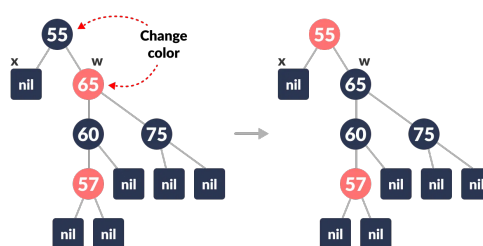
شکل ۱۲: برای حذف ۴۰ ابتدا x را به فرزند راست آن نسبت می‌دهیم.



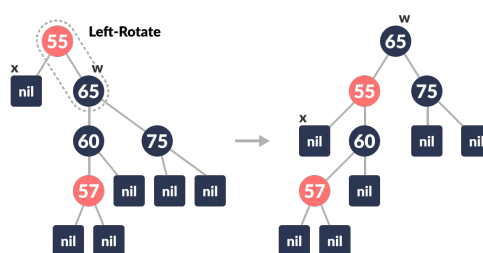
شکل ۱۳: پس از حذف ۴۰، تعداد گره‌های مشکلی در مسیرها نابرابر می‌شوند و باید fixup انجام دهیم.



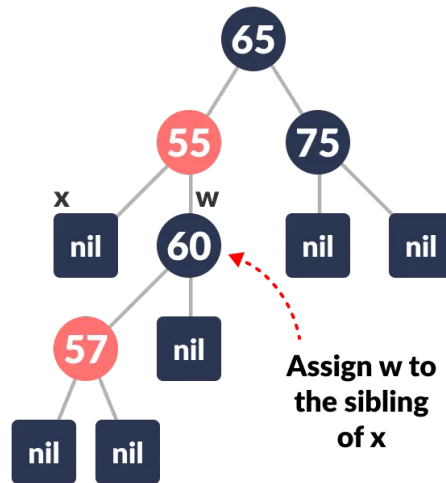
شکل ۱۴: w را به ۶۵ نسبت داده و به عنوان گره برادر/خواهر انتخاب می‌کنیم.



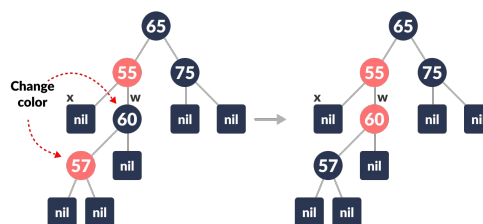
شکل ۱۵: طبق الگوی fixup رنگ ۶۵ و ۵۵ را عوض می‌کنیم.



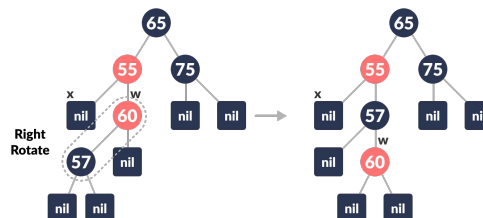
شکل ۱۶: سپس چرخش به چپ حول گره ۵۵ انجام می‌دهیم.



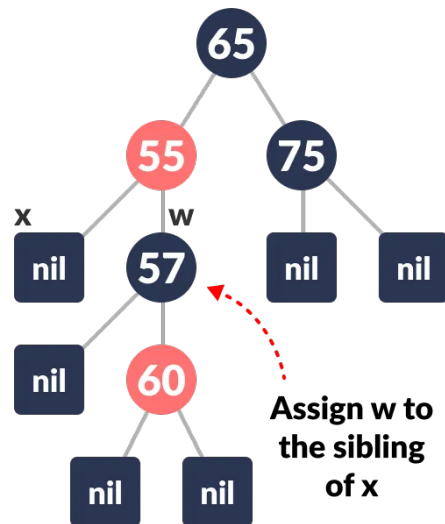
شکل ۱۷: موقعیت فعلی درخت پس از حذف و تعویض. گره ۶۰ به عنوان "w" (برادر) و 'nil' به عنوان "x" (گره دارای دو سیاهی اضافی) مشخص شده است.



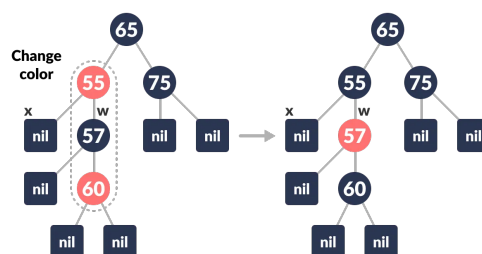
شکل ۱۸: گره ۵۷ از فرزند چپ ۶۰ به فرزند راست ۶۰ منتقل می‌شود و گره ۶۰ رنگ قرمز می‌گیرد.



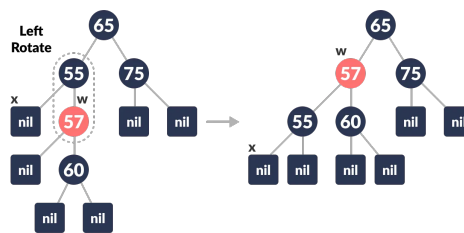
شکل ۱۹: گره ۵۷ جای گره ۶۰ را می‌گیرد و گره ۶۰ به عنوان فرزند راست ۵۷ قرار می‌گیرد. این مرحله شامل تغییر رنگ‌ها برای حفظ خواص قرمز-سیاه است.



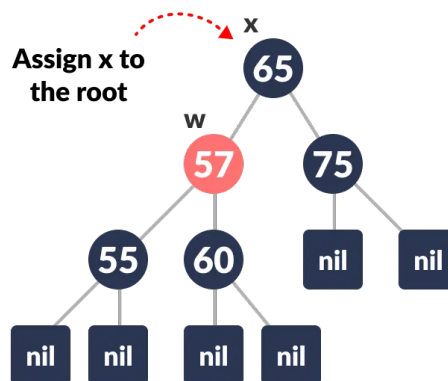
شکل ۲۰: وضعیت پس از حذف گره، که گره ۶۰ به عنوان "x" (نقص دو سیاهی) و گره ۵۷ به عنوان برادر قرمز آن ("sibling") مشخص شده است. (تکرار گام ۷، ولی از زاویه دید x)



شکل ۲۱: چرخش راست در گره ۵۵ انجام می‌شود تا گره ۵۷ به سمت بالا حرکت کند و رنگ‌ها تنظیم شوند.



شکل ۲۲: پس از چرخش و تنظیم رنگ‌ها، گره ۵۷ اکنون ریشه زیردرخت شده است و خواص درخت قرمز-سیاه بازسازی شده‌اند.



شکل ۲۳: شکل نهایی درخت.

۸ منابع

• CLRS (ویرایش چهارم)

• <https://www.programiz.com/dsa/deletion-from-a-red-black-tree>

• <https://medium.com/analytics-vidhya/deletion-in-red-black-rb-tree-92301e1474ea>

• <https://www.cs.csubak.edu/~msarr/visualizations/RedBlack.html>